

# Operating Systems - CSE231

## **Assignment 1**

Brihi Joshi (2016142)  
Taejas Gupta (2016204)

February 15, 2018

## 1. Directory Structure

After downloading the files, the following directory structure is present

```
-2016142_2016204
  -report.pdf
  -Assignment1_2016142_2016204.patch (5.47Kb)
  -sh_task_info.c
  -sh_task_info.h
  -test.c
  -Makefile
```

## 2. Test the Custom System Call

Use the following commands to test the system call -

- Download linux kernel version 3.13.0 in the folder 2016142\_2016204
- Extract the file from linux\_3.13.0-orig.tar.gz using:

```
tar -xf linux_3.13.0.orig.tar.gz
```

- Go to the extracted directory using:

```
cd linux-3.13
```

- Patch the changes to the kernel using:

```
patch -p1 < ../Assignment1_2016142_2016204.patch
```

- make the config files using:

```
make menuconfig
make oldconfig
```

- Now make the changes:

```
make && make modules_install && make install
```

- Go back to the directory where the Assignment was extracted and run the test files:

```
cd ..
make
./test
```

### 3. Description of the code

#### Part One

The first functionality that `sh_task_info()` provides is that given an integer as a parameter, it provides the `task_struct` corresponding to that. We extracted the `pid` struct from the `find_get_pid()` function, which takes the integer `pid` as its argument. It returns a struct `pid` called `pid_struct`. This `pid_struct` is passed as an argument to a function called `pid_task()` which returns a struct `task_struct` called `task`. This is the resultant `task_struct` that we need. In our system call, the fields that we are printing, corresponding to the `task_struct` are :

- Process name
- PID
- Process state
- Priority

We have printed these fields both to `stdout` and to the kernel log message buffer.

#### Part Two

The second functionality that `sh_task_info()` provides is that it stores the fields stated above to the file whose name has been passed to it as an argument. We initially take a buffer string that is used to store our output. We then create a file type struct for the corresponding filename. The position of the pointer in the file is set to 0. Since we need to allocate a free segment of memory to create our file, we use the `get_fs()` function for the same. To open the file, we have given permissions for writing, creating or truncating the contents of the file only. The status of the opened file is stored in `fd`, representing the file descriptor. If the file has been opened successfully, we write the contents of the buffer string into our file using `vfs_write()` function. After writing, we allocate the free segment of memory to the file that we just created.

#### Using the Custom System Call

For calling our system call, we have created a wrapper C program along with its header file, which accepts an integer `pid` and string `filename` as its arguments and invokes the `syscall()` function with the `syscall` number of `sh_task_info` from the `syscall` table as its first argument and `pid` and `filename` as its second and third arguments respectively. We can then use this wrapper C program to use this system call in our userspace. A demonstration for the same has been done in the `test.c` file.

### 4. Inputs Required

Two inputs will be asked while running the userspace program `test.c` -

1. An integer that denotes the PID of the process for which we want to extract the `task_struct` and its fields
2. A string that denotes the filename of the file where the `task_struct` and its fields should be written

## 5. Expected Output and How to interpret it

- If there is no error, the following outputs will be generated:
  1. The process name, PID, state and priority in stdout.
  2. The process name, PID, state and priority in the kernel log, which can be viewed using:  
`dmesg`
  3. The process name, PID, state and priority written to a file for which the name was provided as a parameter.
- If there is an error, the error message will be printed in stdout.

## 6. Error Value and How to interpret them

We have utilized the existing *errno* values that are defined as macros in the kernel.

- **-ENAMETOOLONG** is returned if the file name that is provided is more than 100 characters.
- **-EINVAL** is returned if an invalid PID has been provided. An invalid PID is detected when there is no `task_struct` or process that is associated with it.

For both of these errors, the error statement has been printed in stdout.